

ATTORNEY DOCKET No.

NVIDP044/P000245

U.S. PATENT APPLICATION

FOR A

MULTI-MODE TEXTURE COMPRESSION

ALGORITHM

ASSIGNEE: **NVIDIA** CORPORATION

KEVIN J. ZILKA
PATENT AGENT
P.O. Box 721120
SAN JOSE, CA 95172

NVIDP044/P000245 V1.0

MULTI-MODE TEXTURE COMPRESSION ALGORITHM

RELATED APPLICATION(S)

5

The present application claims the priority of a provisional application filed 8/17/00 under serial number 60/226,240, which is incorporated herein by reference in its entirety for all purposes.

10

FIELD OF THE INVENTION

The present invention relates to computer graphics processing, and more particularly to texture compression algorithms.

15

BACKGROUND OF THE INVENTION

20 In the field of computer graphics, texture mapping is a known technique used to create the appearance of complexity on the surface of rendered objects without actually having to model every detail of the object's surface. Typically, the technique involves mapping a two-dimensional function or array (the texture) onto an object in three-dimensional object space and then projecting the resultant image back to two-dimensional screen space for display. The phrase "texture map" refers to the function or array that is used in the texture mapping process. A common two-dimensional texture map might consist of a repeatable pattern for representing a material, such as wood or marble for example. Three-dimensional texture maps are also used, but not as frequently. Three-dimensional texture maps are usually larger than two-dimensional texture maps. Texture maps are made up of a plurality of numerical values called texels. A texel's numerical value usually corresponds to an RGB color value and perhaps also to an alpha transparency value. (Other
30 parameters may be included in texture maps in addition to, or in lieu of, RGB and

alpha values.) A texel's location within a texture map may be designated using s,t coordinates.

5 A technique known as MIP mapping is also used in texture mapping. MIP mapping involves down-sampling a base texture map numerous times to develop a series of smaller texture maps, each of which represents the base map at a predetermined lower level of resolution. Typically, a map number is assigned to each map. For example, for a system in which two textures were stored, each at four different levels of resolution, eight unique map numbers would be required to refer to the texture maps individually. In systems that use MIP mapping, not only must the base map for each texture be stored in memory, but so must each of the down-sampled maps for each texture. Thus, while texture maps yield important efficiencies for rendering complex images, they can become burdensome in terms of the amount of memory that is required to store them. Indeed, the size of the texture maps used to render an image can in some cases be larger than the rendered image itself.

20 One technique now being used to address the storage problem associated with texture maps is to store the texture maps in the system memory of the host computer rather than in a dedicated texture memory located within the graphics subsystem. This new technique is beneficial to the extent that it eliminates or reduces the need for a large, dedicated texture memory in the graphics subsystem. On the other hand, this new technique also creates a new problem for systems that utilize hardware rendering instead of software rendering: The rendering hardware of the graphics subsystem may make frequent use of the system bus to access large amounts of texture data stored in system memory. This places significant bandwidth demands on both the system bus and system memory.

30 Because of these memory space and bandwidth problems associated with texture mapping, it has become popular to logically partition stored texture maps into a number of equally-sized blocks. This is done because it is usually more

efficient from a bus and memory utilization point of view to retrieve an entire block of texture data from system memory than to retrieve one texel at a time.

For the same reasons, it has also become popular to store texture maps in a compressed format. Various compression algorithms have been used for this purpose including JPEG, run-length encoding, Huffman encoding, vector quantization and Lempel-Ziv compression. Each of these algorithms may be classified in a number of different ways: First, is the algorithm lossy or lossless? Lossy algorithms frequently yield better compression rates than lossless ones, but they do so at the expense of image quality. Second, does the algorithm produce a compression ratio that is fixed or variable? In other words, will the algorithm compress every portion of an image to the same degree, or will it compress highly detailed portions of the image to a lesser degree than other portions of the image? Another factor of importance in choosing compression algorithms is whether and how easily the compressed texture data produced by the algorithm may be accessed randomly. It is often difficult to determine in advance how a given renderer will access a texture. Therefore, the ability to randomly access compressed texture data is extremely beneficial.

Yet another technique that has become popular is a combination of the above-described methods: A texture map may be logically partitioned into blocks, and then compressed one block at a time. If the compressed texture map is stored in system memory as a set of individual compressed blocks, then a desired piece of texture data may be retrieved from system memory by retrieving only the individual compressed block that contains the desired data. Using this technique, the entire compressed texture map does not have to be retrieved from memory simply to access an individual piece of texture data within it. Moreover, because the block is retrieved in compressed format, additional bus and memory bandwidth savings are realized.

One difficulty that arises when applying the foregoing methods is that of selecting an appropriate compression format for a particular block. Certain compression algorithms work better with certain types of data. For example, certain textures lend themselves to more effective compression using certain texture compression algorithms due to the specific colors or various other aspects associated with the texture data. To date, conventional texture compression algorithms apply the same compression algorithm to all blocks of texture data, regardless of the various characteristics of the texture data.

10

11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217

DISCLOSURE OF THE INVENTION

A multi-mode texture compression algorithm is provided for effective compression and decompression texture data during graphics processing. Initially, a request is sent to memory for compressed texture data. Such compressed texture data is then received from the memory in response to the request. At least one of a plurality of compression algorithms associated with the compressed texture data is subsequently identified. Thereafter, the compressed texture data is decompressed in accordance with the identified compression algorithm.

5 Prior to sending the request, the texture data may be compressed utilizing all of the compression algorithms. The most favorable compressed texture data is then selected. As an option, the most favorable compressed texture data may be the most accurate replication of an original version of the texture data. Next, the most favorable compressed texture data is stored in the memory.

10 In one embodiment, a mode identifier may be stored with the compressed texture data. Moreover, the compression algorithm associated with the compressed texture data may be identified utilizing the mode identifier. Optionally, the mode identifier may include at least one mode bit.

15 Various specific compression algorithms may be utilized in the context of the present embodiment. For example, at least one of the compression algorithms may represent a 4x4 block of texels of the texture data utilizing two bits per texel if the texels are opaque. Further, each 4x4 block of texels may include two 16-bit colors stored in an RGB 565 format and two additional colors created by
20 interpolating between the two 16-bit colors stored in the RGB 565 format to form a 4-entry lookup table. A 2-bit index may be adapted for being used to determine which 16-bit color from the lookup table is used for each texel of the 4x4 block of texels. Moreover, transparent texels may be represented by making one of the four 16-bit colors transparent.

Still yet, at least one of the compression algorithms may represent a 4x8 block of texels utilizing three bits per texel. Each 4x8 block of texels may include two 15-bit colors stored in an RGB 555 format and five additional colors created by interpolating between the two 15-bit colors stored in the RGB 555 format to form an 8-entry lookup table. An eighth 15-bit color may be defined to be a transparent color. Further, a 3-bit index may be used to determine which 15-bit color from the lookup table is used for each texel in the 4x8 block of texels.

In still another embodiment, at least one of the compression algorithms may represent a 4x8 block of texels utilizing two bits per texel if the textels are opaque. Each 4x8 block of texels may include four 15-bit colors in an RGB 555 format to form a 4-entry lookup table. A 2-bit index may be adapted for being used to determine which of the four 15-bit colors is assigned to each texel.

In still yet another embodiment, at least one of the compression algorithms may represent a 4x8 block of texels with two bits per texel. Each 4x8 block of texels may include three 20-bit colors stored in a 5555 format: 5 bits for each of red, green, blue, and alpha (opacity). A first and second one of the 20-bit colors may be used for primary colors of a left 4x4 sub-block of the 4x8 block of texels. Further, a second and third one of the colors may be used for primary colors of the right 4x4 sub-block of the 4x8 block of texels. Two additional 20-bit colors may be created in each 4x4 sub-block of texels by interpolating between the 20-bit colors associated with the corresponding 4x4 sub-block of texels. A 2-bit index may be adapted for being used to determine which of the four 20-bit colors is assigned to each texel. Further, a lookup table may be used to determine which 20-bit color is applied to each texel.

These and other advantages of the present invention will become apparent upon reading the following detailed description and studying the various figures of the drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

5 The foregoing and other aspects and advantages are better understood from the following detailed description of a preferred embodiment of the invention with reference to the drawings.

10 Figure 1 illustrates an exemplary architecture in which the present embodiment may be implemented.

Figure 2 illustrates a particular framework associated with the graphics processor of the exemplary architecture of Figure 1.

15 Figure 3 shows the manner in which texture data is compressed and stored in memory for use by the foregoing architecture.

Figure 4 illustrates a texture data compression method for use during graphics processing, in accordance with one embodiment.

20

DESCRIPTION OF THE PREFERRED EMBODIMENTS

Figure 1 illustrates an exemplary architecture **100** in which the present
5 embodiment may be implemented. As shown, a central processing unit **102** is
coupled to a chipset **104** which is, in turn, coupled to memory **106**. In one
embodiment, this memory **106** may take the form of an array of one or more
dynamic RAM memory chips, or another type of storage device capable of storing
data. In use, the memory **106** may be used to store large amounts of compressed
10 texture data in a manner that will soon be set forth. Such compressed texture data
is accessible by the chipset **104** under the control of the central processing unit
102.

Coupled to the chipset **104** via a bus **105** is a graphics subsystem **107**
15 including a graphics processor **108**. A frame buffer/texture memory **110** is coupled
to the graphics processor **108** for populating a display **112** coupled thereto.

As will soon become apparent, compressed texture data may be stored in
the memory **106**, memory associated with the graphics subsystem **107**, or any other
20 memory, such as a CD-ROM or other disk drive. Such compressed texture data
may further be compressed utilizing various compressed algorithms based on
which format provides the most favorable results.

In use, the graphics processor **108** is capable of retrieving the compressed
25 texture data, and identifying the particular compression format associated with the
compressed texture data. The compressed texture data may then be decompressed
for the purpose of texture mapping. By optimizing the compression and
decompression of the texture data, the present embodiment decreases the
bandwidth required to transfer the texture data and further increases the fill-rate
30 during texture mapping.

Figure 2 illustrates a particular framework 200 associated with the graphics processor 108 of the graphics subsystem 107 of Figure 1, in accordance with one embodiment. As shown, a texture fetch module 202 is provided for sending a request for compressed texture data to the frame buffer/texture memory 110, and receiving the compressed texture data from the frame buffer/texture memory 110. In one embodiment, 128-bits of compressed texture data may be requested and retrieved from the frame buffer/texture memory 110.

Coupled to the frame buffer/texture memory 110 is a format detection module 204. In use, the format detection module 204 is adapted for identifying at least one of a plurality of compression algorithms associated with the compressed texture data requested from the frame buffer/texture memory 110. More information on the various specific compression algorithms will be set forth hereinafter in greater detail.

A plurality of decompression modules 206 are coupled between the texture fetch module 202 and the format detection module 204. The decompression modules 206 are adapted for decompressing the compressed texture data in accordance with the compression algorithm identified by the format detection module 204. While not shown, it should be noted that the graphics processor 108 and/or other components of the graphics subsystem 107 may include additional conventional functionality such as transform, lighting, etc.

Figure 3 illustrates the manner in which texture data is compressed and stored in memory for use by the foregoing architecture. As mentioned earlier, compressed texture data may be stored in the memory 106, memory associated with the graphics subsystem 107, or any other memory.

In use, uncompressed texture data is initially received in operation 302 for compression purposes. In one embodiment, the texture data may be received in blocks. For example, the texture data may be received in 4x8 or 4x4 blocks.

It should be noted that the blocks of texture data may be compressed on or off the architecture **100** of Figure **1**. In one embodiment, the texture data may be compressed on a CD-ROM and read by the architecture **100** of Figure **1**. In another embodiment, the texture data may be compressed by the central processing unit **102** and immediately stored on the architecture **100** of Figure **1**. Of course, the texture data may be compressed in any desire manner and in any particular environment.

As shown in Figure **3**, the uncompressed texture data is compressed by a plurality of compression algorithms in operations **304a-d**. Thereafter, in operations **306a-d**, the compressed texture data is decompressed utilizing a plurality of de-compression algorithms that are complimentary to the compression algorithms in operations **304a-d**.

The output of the operations **306a-d** is then compared to the original uncompressed texture data in operations **308a-d** for determining an amount of error. In one embodiment, such error may take the form of a distance metric. It should be understood that the lowest amount of error reflects the most favorable compression algorithm. In various other embodiments, the most favorable compression algorithm may be gauged utilizing any desired criteria such as compression size, etc. The results of such comparison are then used by a control module **310** for controlling a multiplexer **312** to store the most favorable results to memory in operation **314**.

The compressed texture data is then ready for being accessed by the graphics processor **108** for an improved decompression algorithm. For reasons that will soon become apparent, each of the compression algorithms is capable of storing particular mode bits with the compressed texture data for the purpose of identifying the compression algorithm. More information on the various specific compression algorithms will be set forth hereinafter in greater detail.

Figure 4 illustrates a texture data compression method 400 for use during graphics processing. Initially, in operation 402, a request is sent to memory for compressed texture data. In one embodiment, such request may be made by a texture fetch module 202 like that shown in Figure 2. Of course, however, the request may be made by the central processing unit 102 of Figure 1, another module associated with the graphics subsystem 107 or any logic for that matter.

Next, in operation 404, such compressed texture data is received from the memory in response to the request. At least one of a plurality of compression algorithms associated with the compressed texture data is subsequently identified. This may be accomplished by determining mode bits associated with the compressed texture data in decision 405. In one embodiment, such compressed texture data may be received by the format detection module 204 like that shown in Figure 2. Of course, however, the compressed texture data may be received by the central processing unit 102 of Figure 1, another module associated with the graphics subsystem 107 or any logic for that matter.

Thereafter, the compressed texture data is decompressed in accordance with the identified compression algorithm in operations 406a-d. Various specific decompression algorithms may be utilized in the context of the present embodiment. Each of such decompression algorithms is a compliment of a particular compression algorithm. In one embodiment, the decompression may be accomplished by the decompression modules 206 like those shown in Figure 2. Of course, however, the decompression may be accomplished by the central processing unit 102 of Figure 1, another module associated with the graphics subsystem 107 or any logic for that matter.

Four examples of the aforementioned compression algorithms will now be set forth. It should be noted that more or less various other compression algorithms may be utilized per the desires of the user.

In one embodiment, each of the formats compress an 8x4 texel blocks into 128 bits. During the compression phase, one of the four formats for each block is selected based on which encoding scheme results in the best overall visual quality.

- 5 Unused pixel locations along the right or bottom edges within a block may contain a repetition of the values in used locations. The total size of an image is $\text{ceil}(\text{width}/8) * \text{ceil}(\text{height}/4) * 16$ bytes.

- 10 In each compression format, the 32 texels of the 8x4 block are partitioned into two 4x4 sub-blocks according to the diagram shown in Table 1.

Table 1

15	t0	t1	t2	t3	t16	t17	t18	t19
	t4	t5	t6	t7	t20	t21	t22	t23
	t8	t9	t10	t11	t24	t25	t26	t27
	t12	t13	t14	t15	t28	t29	t30	t31

- 20 By way of summary, a first one of the compression algorithms may represent a 4x8 block of texels utilizing three bits per texel. Each 4x8 block of texels may include two 15-bit colors stored in an RGB 555 format and five additional colors created by interpolating between the two 15-bit colors stored in the RGB 555 format to form an 8-entry lookup table. An eighth 15-bit color may be defined to be a transparent color. Further, a 3-bit index may be used to
- 25 determine which 15-bit color from the lookup table is used for each texel in the 4x8 block of texels. The present compression algorithm works well when colors are “peppered” about in an image, and or generally unorganized. Further, more area is covered by the present compression algorithm. Thus, the present compression algorithm is ideal for spatial resolution.

30

A second one of the compression algorithms may represent a 4x8 block of texels utilizing two bits per texel if the texels are opaque. Each 4x8 block of texels

may include four 15-bit colors in an RGB 555 format to form a 4-entry lookup table. A 2-bit index may be adapted for being used to determine which of the four 15-bit colors is assigned to each texel. Thus, the present compression algorithm is ideal for complex color areas.

5

A third one of the compression algorithms may represent a 4x4 block of texels of the texture data utilizing two bits per texel if the texels are opaque. Further, each 4x4 block of texels may include two 16-bit colors stored in an RGB 565 format and two additional colors created by interpolating between the two 16-bit colors stored in the RGB 565 format to form a 4-entry lookup table. A 2-bit index may be adapted for being used to determine which 16-bit color from the lookup table is used for each texel of the 4x4 block of texels. Moreover, transparent texels may be represented by making one of the four 16-bit colors transparent.

10

15

Still yet, a fourth one of the compression algorithms may represent a 4x8 block of texels with two bits per texel. Each 4x8 block of texels may include three 20-bit colors stored in a 5555 format. A first and second one of the 20-bit colors may be used for primary colors of a left 4x4 sub-block of the 4x8 block of texels. Further, a second and third one of the colors may be used for primary colors of the right 4x4 sub-block of the 4x8 block of texels. Two additional 20-bit colors may be created in each 4x4 sub-block of texels by interpolating between the 20-bit colors associated with the corresponding 4x4 sub-block of texels. A 2-bit index may be adapted for being used to determine which of the four 20-bit colors is assigned to each texel. Further, a lookup table may be used to determine which 20-bit color is applied to each texel. The present compression algorithm is ideally suited for situations where colors span across the screen, blending with other colors.

20

25

30

When the texture data is compressed, a mode identifier (i.e. 2-bit field) is stored in each block and is used to determine which of the four foregoing

compression schemes was utilized for best visual quality. Depending on which algorithm is used for a given block, the proper decompression logic is applied to generate decoded 32-bit texels which can then be used by texture mapping hardware of the graphics subsystem.

5

More information will now be set forth regarding the format associated with each of the foregoing four compression algorithms.

First Compression Format (CC_HI)

10

Table 2 summarizes the first compression format.

Table 2

15	bit 127		(rgb555)		(3-
	bit/texel)				
	<u>mode[1:0]</u>	<u>color1</u>	<u>color0</u>	<u>texel 31 to 16</u>	<u>texel 15 to</u>
	<u>0</u>				
	2	15	15	48	48
20					
		[127:126]	mode[1:0]		
		[125:121]	red of color1		
		[120:116]	green of color1		
25		[115:111]	blue of color1		
		[110:106]	red of color0		
		[105:101]	green of color0		
		[100:96]	blue of color0		
		[95:93]	texel 31		
30		...			
		[50:48]	texel 16		
		[47:45]	texel 15		
		...			
		[2:0]	texel 0		

35

In the first format, mode = 00b, the 15-bit color1 (RGB555 format) and color0 (RGB555 format) colors are converted into 24-bit RGB888 colors by duplicating the upper 3 bits for the 3 LSBs. The 24-bit converted color1 and color0 are then used to linearly interpolate 5 more levels of color to create seven total levels of colors and 1 alpha (transparent) color. The first seven colors have alpha=ffh (opaque), while the eighth color is defined to be transparent black (r,g,b=00h, alpha=00h).

These eight 32-bit colors are used as the contents of an 8-entry (3 bit index) lookup table. For all 32 texels in the block, each texel's 3-bit index value is used to index the lookup table, the output from the lookup table representing the 32-bit color (ARGB8888) for that texel.

Table 3 illustrates the manner in which RGB888 colors are generated from RGB555 colors.

Table 3

Color1 (red) =	{ [125:121], [125:123] }
Color1 (green) =	{ [120:116], [120:118] }
Color1 (blue) =	{ [115:111], [115:113] }
Color0 (red) =	{ [110:106], [110:108] }
Color0 (green) =	{ [105:101], [105:103] }
Color0 (blue) =	{ [100:96], [100:98] }

Table 4 illustrates the manner in which the seven ARGB8888 colors are created from two RGB888 colors (operations performed individually for each color channel).

Table 4

Color[0] = color0[r,g,b],	alpha[0] = ffh
Color[1] = (5*color0[r,g,b] + color1[r,g,b] + 3) / 6	alpha[1] = ffh
Color[2] = (4*color0[r,g,b] + 2*color1[r,g,b] + 3) / 6	alpha[2] = ffh

Color[3] = (3*color0[r,g,b] + 3*color1[r,g,b] + 3)/6 alpha[3] = ffh
 Color[4] = (2*color0[r,g,b] + 4*color1[r,g,b] + 3)/6 alpha[4] = ffh
 Color[5] = (color0[r,g,b] + 5*color1[r,g,b] + 3)/6 alpha[5] = ffh
 Color[6] = color1[r,g,b], alpha[6] = ffh
 Color[7] = where r,g,b = 00h, alpha[7]=00h

Table 5 illustrates the table lookup associated with Table 4.

Table 5

10	3-bit index of <u>texel31 to texel0</u>	Color for texel 31 to texel 0 <u>(ARGB8888)</u>
	0	color[0] =>
15	{a[7:0],r[7:0],g[7:0],b[7:0]}	
	1	color[1]
	2	color[2]
	3	color[3]
	4	color[4]
20	5	color[5]
	6	color[6]
	7	color[7]

Second Compression Format (CC_CHROMA)

25 Table 6 summarizes the second compression format.

Table 6

30	bit 127 (rgb555)	(2-bit/texel)
	<u>Mode[2:0]</u> <u>unused</u> <u>color3</u> <u>color2</u> <u>color1</u> <u>color0</u> <u>texel 31 to 16</u> <u>texel 15 to 0</u>	
	3 1 15 15 15 15 32 32	
35	[127:125] mode[2:0]	
	[124] unused	
	[123:119] color3 (x5)	
	[118:114] color3 (g5)	
	[113:109] color3 (b5)	
	[108:104] color2 (x5)	
40	[103:99] color2 (g5)	

5

10

25

30

35

NVIDP044/P000245 V2.0

Table 8 summarizes the third compression format.

Table 8

5

bit 127	(rgb555)				(2-bit/texel)			
<u>mode[0]</u>	<u>gLSB[1:0]</u>	<u>alpha[0]</u>	<u>color3</u>	<u>color2</u>	<u>color1</u>	<u>color0</u>	<u>texel 31to16</u>	<u>texel 15to0</u>
1	2	1	15	15	15	15	32	32

10

[127] mode[0]
[126:125] gLSB[1:0] (LSBs of green for color 1 & color

3)

[124] alpha[0]

15

[123:119] color3 (r5)

[118:114] color3 (g5)

[113:109] color3 (b5)

[108:104] color2 (r5)

[103:99] color2 (g5)

20

[98:94] color2 (b5)

[93:89] color1 (r5)

[88:84] color1 (g5)

[83:79] color1 (b5)

[78:74] color0 (r5)

25

[73:69] color0 (g5)

[68:64] color0 (b5)

[63:62] texel 31

...

[33:32] texel 16

30

[31:30] texel 15

...

[1:0] texel 0

In the third format, mode[0]=1 (only one bit), color2 and color3 are used
35 for texels 31 to 16, and color0 and color1 are used for texels 15 to 0. When
alpha[0] = 0, the two pairs of colors (colors 0 and 1 for texels 15 to 0 and colors 2
and 3 for texels 31 to 16) are interpreted as 16-bit RGB565 colors. For color1 and
color3, the LSB (bit 0) of the green channel comes from the gLSB bits
(color1.green[0] = bit 125, color3.green[0] = bit 126). For color0 and color2, the

LSB (bit 0) of the green channel comes from the upper select bit for texel 0 and texel 16, respectively ($\text{color0.green}[0] = \text{bit 1 xor bit 125}$, $\text{color2.green}[0] = \text{bit 33 xor bit 126}$). The two 16-bit colors are then expanded to a 24-bit RGB888 format by bit replication (most significant bits replicated in the least significant bits), and
5 are then used to create 2 more levels of color in between the color0/2 and color1/3 values through linear interpolation. A total of 4 colors are therefore available for 2-bit index per texel selection.

When $\text{alpha}[0]=1$, color0 and color2 are interpreted as 15-bit RGB555
10 colors, and color 1 and color3 are interpreted as RGB565 colors. For color0 and color2 , the 15-bit RGB555 colors are expanded to 24-bit RGB888 colors by bit replication. For color1 and color3 , the LSB (bit 0) of the green channel comes from the gLSB bits ($\text{color1.green}[0] = \text{bit 125}$, $\text{color3.green}[0] = \text{bit 126}$), and then bit replication is used to convert from the 16-bit RGB565 format to a 24-bit RGB888
15 format. A third color is created by linear interpolation (interpolating between the converted 24-bit RGB888 color0 and color1 for texels 15 to 0, and interpolating between the converted 24-bit RGB888 color2 and color3 for texels 31 to 16).

A fourth color (texel index 0x3) is defined to be transparent black
20 (r,g,b=00h, alpha=00h). A total of 4 colors are therefore available for 2-bit index per texel selection. The 32-bit (ARGB8888) color value for all texels is obtained by performing a table lookup using each texel's 2-bit index.

Table 9 illustrates the manner in which the 24-bit (RGB888) base colors
25 color3 and color2 are created.

Table 9

30 $\text{Color3}(\text{red}) = \{ [123:119], [123:121] \}$
 $\text{Color3}(\text{green}) = \{ [118:114], [126], [118:117] \}$
 $\text{Color3}(\text{blue}) = \{ [113:109], [113:111] \}$
 $\text{Color2}(\text{red}) = \{ [108:104], [108:106] \}$

```
Color2(green) = (alpha[0]=1) ?  
{ [103:99], [103:101] }  
  
    { [103:99], [33]^ [126], [103:102] }  
5    Color2(blue) = { [98:94], [98:96] }
```

Table 10 illustrates the manner in which the 24-bit (RGB888) base colors color1 and color0 are created.

10 Table 10

```
Color1(red) = { [93:89], [93:91] }  
Color1(green) = { [88:84], [125], [88:87] }  
Color1(blue) = { [83:79], [83:81] }  
15    Color0(red) = { [78:74], [78:76] }  
Color0(green) = (alpha[0]=1) ? { [73:69], [73:71] }  
    : { [73:69], [1]^ [125],  
[73:72] }  
Color0(blue) = { [68:64], [68:66] }
```

20

When alpha[0]=0, because one of the texel select bits is used to determine a bit of color0 and color2, the compressor may have to perform some very tricky operations. Table 11 illustrates the method as to how to generate color0 and color1 and the associated select bits (the same method applies to determining the LSB of green for color2 and color3).

25

Table 11

1. Determine the 16-bit RGB565 color values for color0 & color1.
2. Determine the select bits for each pixel in the 4x4 sub-block.
3. If (pixel[0].select[1] != color0.green[0]^color1.green[0]) then swap color0 & color1, and invert all the select bits.

35

Table 12 shows psuedo-C code to generate bits 0-31, bits 64-93 & bit 125 based on the initial color0, color1 and pixel indices.

Table 12

```
5      struct RGB565 {Byte red; Byte green; Byte blue};

      struct CSels {Byte index[16]};

10      // cc_mixed_right_half derives bits[93:64] of the 128
      bit data word of a
      // CC_MIXED non-alpha compression block and returns them
      in 'bits_64_to_31'.
      // Plus, as a bonus, you will receive bit 125,
15      containing the LSB of
      // the green channel of color1, and bits_0_to_31,
      containing all of the pixel indices.
      void
      cc_mixed_right_half( RGB565 color0, RGB565 color1,
20                          CSels pix,
                          Dword &bits_0_to_31,
                          Dword &bits_64_to_93,
                          Bit &bit125)
      {
25          RGB565 o_color0;
          RGB565 o_color1;

          // Determine if we need to switch color0 & color1
          if (((pix.index[0] >> 1) & 1) != ((color0.green ^
30      color1.green) & 1)) {
              o_color1 = color0;
              o_color0 = color1;

              for (int i=0; i<16; i++)
35                  pix.index[i] = ~pix.index[i] & 3;
              } else {
                  o_color0 = color0;
```

```
        o_color1 = color1;
    }

    // Save LSB of color1.green in bit125
5    bit125 = o_color1.green & 1;

    // Convert color0 & color1 to RGB555, and then munge
into bits 64 to 93
    o_color0.green >>= 1;
10    o_color1.green >>= 1;

    bits_64_to_93 = ( (o_color1.red<<25) |
(o_color1.green<<20) | (o_color1.blue<<15)
                    | (o_color0.red<<10) | (o_color0.green<<5)
15    | (o_color0.blue) );

    // Munge the pixel indices into bits 0 to 31
    bits_0_to_31 = 0;

20    for (int i=0; i<16; i++)
        bits_0_to_31 |= pix.index[i]<<(i*2);
    }
```

Table 13 shows the manner in which the 4-entry lookup table for texels 31
25 to 16 is generated.

Table 13

```
    If alpha[0]=0,
30    Color[0] = color2[r,g,b] , alpha=ffh
    Color[1] = (2 * color2[r,g,b] + color3[r,g,b] + 1)
/ 3, alpha=ffh
    Color[2] = (color2[r,g,b] + 2 * color3[r,g,b] +1)
/ 3, alpha=ffh
35    Color[3] = color3[r,g,b] , alpha=ffh

    If alpha[0]=1,
```

```

Color[0] = color2[r,g,b] , alpha=ffh
Color[1] = (color2[r,g,b] + color3[r,g,b]) / 2,
alpha=ffh
Color[2] = color3[r,g,b] , alpha=ffh
5 Color[3] = [a,r,g,b] = 00h

```

Table 14 shows the manner in which the 4-entry lookup table for texels 15 to 0 is generated.

10 Table 14

```

If alpha[0]=0,
Color[0] = color0[r,g,b] , alpha=ffh
Color[1] = (2 * color0[r,g,b] + color1[r,g,b] + 1)
15 / 3, alpha=ffh
Color[2] = (color0[r,g,b] + 2 * color1[r,g,b] + 1)
/ 3, alpha=ffh
Color[3] = color1[r,g,b] , alpha=ffh

20 If alpha[0]=1,
Color[0] = color0[r,g,b] , alpha=ffh
Color[1] = (color0[r,g,b] + color1[r,g,b]) / 2,
alpha=ffh
Color[2] = color1[r,g,b] , alpha=ffh
25 Color[3] = [a,r,g,b] = 00h

```

Table 15 illustrates the resultant table lookup.

Table 15

```

30 2-bit index of          Color for texel 31 to texel 0
texel 31 to texel 0  ARGB8888

0          color[0], {a[7:0], r[7:0], g[7:0], b[7:0]}
35 1          color[1]
2          color[2]

```


3 color[3]

Fourth Compressed Texture Format (CC_ALPHA)

5 Table 16 summarizes the fourth compression format.

Table 16

10	bit 127	(argb5555)			(2-bit/texel)	
	mode[2:0]	lerp	alpha2	alpha1	alpha0	color2 color1 color0 texel 31 to 16 texel 15 to
	0					
	3	1	5	5	5	15 15 15 32 32
15				[127:125]	mode[2:0]	
				[124]	lerp	
				[123:119]	color2(a5)	
				[118:114]	color1(a5)	
				[113:109]	color0(a5)	
20				[108:104]	color2(r5)	
				[103:99]	color2(g5)	
				[98:94]	color2(b5)	
				[93:89]	color1(r5)	
				[88:84]	color1(g5)	
25				[83:79]	color1(b5)	
				[78:74]	color0(r5)	
				[73:69]	color0(g5)	
				[68:64]	color0(b5)	
				[63:62]	texel 31	
30				...		
				[33:32]	texel 16	
				[31:30]	texel 15	
				...		
				[1:0]	texel 0	

35

In the fourth format, mode[2:0]=011b, three 20-bit colors color2, color1 and color0 (ARGB5555) are converted to a 32-bit (ARGB8888) format by

duplicating the upper 3-bits for the 3 LSBs (all the color channels and the alpha channel are converted from 5-bit formats to 8-bit formats using this bit duplication).

- 5 Table 17 illustrates the manner in which the 32-bit (RGB8888) base colors color2, color1, and color0 are created.

Table 17

10	Color2(alpha) = { [123:119], [123:121] }
	Color2(red) = { [108:104], [108:106] }
	Color2(green) = { [103:99], [103:101] }
	Color2(blue) = { [98:94], [98:96] }
	Color1(alpha) = { [118:114], [118:116] }
15	Color1(red) = { [93:89], [93:91] }
	Color1(green) = { [88:84], [88:86] }
	Color1(blue) = { [83:79], [83:81] }
	Color0(alpha) = { [113:109], [113:111] }
	Color0(red) = { [78:74], [78:76] }
20	Color0(green) = { [73:69], [73:71] }
	Color0(blue) = { [68:64], [68:66] }

- 25 When lerp = 0 (bit 124 = 0), the converted 32-bit colors color2, color1, and color0 are used directly as the first 3 entries in the 4-entry lookup table. The last entry in the 4-entry lookup table, accessed with index=3, is defined to be transparent black (rgb=00h, alpha=00h). A total of 4 colors are therefore available for 2-bit index per texel selection, and the 32-bit (ARGB8888) color value for all texels is obtained by performing table lookup using each texel's 2-bit index.

- 30 Table 18 illustrates the table lookup (when lerp = 0).

Table 18

Index of texel 31 to 0 Color for texel 31 to texel 0

(ARGB8888)

	0	Color[0] = color0 alpha = alpha0
	1	Color[1] = color1 alpha = alpha1
5	2	Color[2] = color2 alpha = alpha2
	3	Color[3] = 000000h alpha = 00h

When lerp = 1 (bit 124 = 1), the converted 32-bit colors color2 and color1 are used as the 32-bit base colors for texels 31 to 16, and the converted 32-bit colors color1 and color0 are used as the base colors for texels 15 to 0. The 32-bit base colors are then used to create 2 more levels of color through linear interpolation. A total of 4 colors are therefore available for 2-bit index per texel selection, and the 32-bit (ARGB8888) color value for all texels is obtained by performing table lookup using each texel's 2-bit index.

15

Table 19 illustrates the manner in which the 4 colors used in the 4-entry lookup table are created from the 32-bit base colors (when lerp = 1).

Table 19

20

For texel 31 to texel 16
Color[0] = color2[a,r,g,b]
Color[1] = (2 * color2[a,r,g,b] + color1[a,r,g,b] + 1) /
3
Color[2] = (color2[a,r,g,b] + 2 * color1[a,r,g,b] + 1) /
3
Color[3] = color1[a,r,g,b]

25

3

For texel 15 to texel 0

30

Color[0] = color0[a,r,g,b]
Color[1] = (2 * color0[a,r,g,b] + color1[a,r,g,b] + 1) /
3
Color[2] = (color0[a,r,g,b] + 2 * color1[a,r,g,b] + 1) /
3
Color[3] = color1[a,r,g,b]

3

3

35

Table 20 illustrates the table lookup (when lerp = 1).

Table 20

5	<u>Index of texel 31 to 0</u>	<u>Color for texel 31 to texel 0</u> <u>ARGB8888)</u>
	0	color[0]
	1	color[1]
10	2	color[2]
	3	color[3]

The present embodiment thus creates images with higher quality than other texture compression schemes by using multiple compression techniques for each texture. This allows the compressor to be more accurate in reproducing specific portions of an image and/or different types of images as the best possible technique is applied to each texel block.

Further, unlike prior art compression schemes which use an 8-bit compression format when compressing textures with multi-bit alpha components (alpha is used for transparency information), the present embodiment is capable of using a 4-bit format for a better compression ratio. As a result, the compression ratio of the present compression algorithm is twice that of the prior art algorithms when compressing 16 or 32-bit textures which include alpha information. This substantially increases the number of textures which can be stored in a given amount of memory, and also reduces the amount of bandwidth required for texturing.

While various embodiments have been described above, it may be understood that they have been presented by way of example only, and not limitation. Thus, the breadth and scope of a preferred embodiment may not be limited by any of the above described exemplary embodiments, but may be defined only in accordance with the following claims and their equivalents.